PCP

The Parallel C Preprocessor

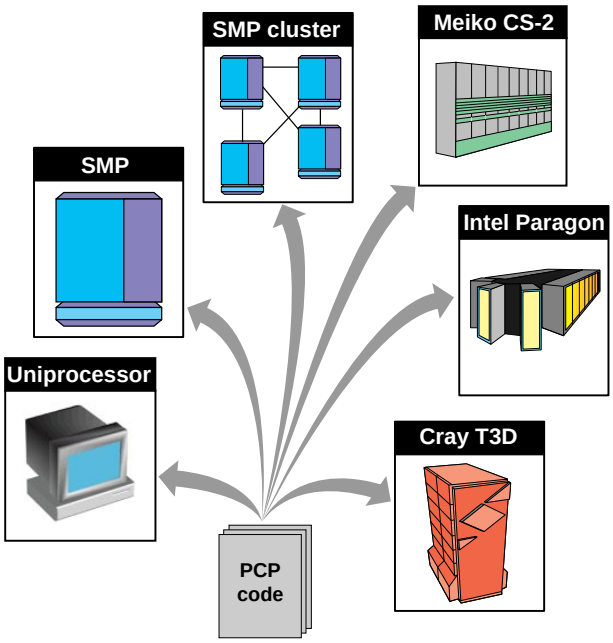
University of California
Lawrence Livermore
National Laboratory

Mission

PCP, the Parallel C Preprocessor, is a programming model that provides a simple yet powerful shared-memory C parallel programming language for uniprocessor, symmetric multiprocessor (SMP), and distributed memory massively parallel processing architectures.

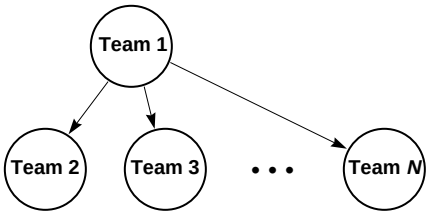
Impact

PCP provides a means to solve the problem of maintaining multiple versions of a code for multiple target architectures. The PCP code runs on hardware ranging from a uniprocessor to a cluster of SMPs to a teraflop supercomputer with distributed memories. PCP is a low-level programming model that allows the user to code all types of scientific algorithms, including irregular calculations. The user can explicitly manage data layout and communication via shared memory, thus avoiding the high overhead commonly associated with message passing.



PCP runs on hardware ranging from a uniprocessor to a cluster of SMPs to a teraflop supercomputer with distributed memories.

PCP is a simple shared-memory parallel C programming model that allows the user to program for a wide range of parallel hardware architectures. In the PCP programming model there is one thread of control per physical processor; these processors form a team. Each processor executes the code from start to finish. The initial group may be subdivided into subteams for nested parallel tasks.



PCP gives the user a global as well as a local view of memory. PCP adds to the ANSI standard C language a few powerful keywords that express parallelization, synchronization, and data placement. PCP provides the ANSI C-style type qualifier keywords **`shared`**, **`private`**, and **`teamprivate`**. These make it possible to specifically designate data location directly or through the use of pointers. These constructs may also be applied to subteams of the initial processor team to address nested task parallelization. By proper use of these type qualifiers, the same PCP code can take advantage of local memory bandwidth on distributed memory machines and shared memory and caching on an SMP.

For example,

```
shared int shared *ptr;
```

is a pointer in shared memory that points to an integer also in shared memory. On a distributed memory computer, the elements of an array declared with the type qualifier **`shared`** are automatically spread across the memory banks in a cyclic manner. The PCP language translator takes care of the addressing and communication. Direct or pointer references to an array element are translated to code that accesses the memory location on the processor where the data resides.

These type qualifiers are reducible, i.e., have no overhead, on architectures for which there is no hardware support for separate shared and local memories, specifically SMPs and uniprocessors.

PCP Syntax

master

The processor whose current team index is 0 executes the code inside a **`master`** block. A **`master`** block is often used for initialization as well as input, output, and memory allocation. The **`master`** blocks are used to initialize shared data, such as accumulators, that all team members will access.

```
master {
  <declarations>
  <executable code>
}
```

forall

The **forall** loop is the PCP concurrent equivalent of the C language **for** loop. It achieves a fine-grained parallelism by dividing up the iterations of the **for** loop among the members of the processor team:

```
forall
  (int i = <start>;
   <cond>; i += <step>) {
  work(i)
}
```

By default the indices of the loop are interleaved among the members of the executing team. The loop index variable must be declared in the **forall** statement.

Synchronization

Each of the processors of a team of processors executes the code at its own rate unless explicit synchronization primitives are encountered. One basic and frequently used form of synchronization is the **barrier**:

```
barrier;
```

A barrier requires all members of the team to arrive at the barrier before any are allowed to continue. Each subteam has its own distinct barrier. A barrier is often used after a **master** block, or a **forall** loop, to ensure that the preceding work is complete before any processor is allowed to continue.

Locks

Concurrency must be inhibited in a statement that reads, modifies, and then writes a variable that many processors are modifying. To prevent processors from destructively interfering with each other, PCP restricts entrance to a critical section of a code so that only one processor may execute it at a time. This is accomplished by using a lock.

PCP offers spin wait locks that are implemented by variables of the **lock** data type, which has the two states, **locked** and **unlocked.** A **lock** variable is a statically allocated and initialized C data type:

```
lock var = unlocked;
```

Functions that change the state of a lock are **lock()** and **unlock()**, which take the pointer to the lock variable as an argument. **lock()** waits until the **lock** is **unlocked** and then atomically sets it to **locked**. **unlock** sets it to **unlocked**. For example:

```
lock(&var);
<critical section>
unlock(&var);
```

Processor Teams

Static team splitting is used to divide a number of tasks, known at compile time, among subteams that are split from the parent:

```
split [weight1]
  { <task1> }
and [weight2]
  { <task2> }
...
and [weightn]
  { <taskn> }
```

The tasks may be executed in any order, including sequentially if the team encountering the **split** statement cannot be split for some reason. If one task contains more work than another, one may assign weights to the blocks of work to achieve load balancing. The weights determine the fraction of the current team's processors that are split into each subteam. Weights are computed at run time.

The dynamic or loop-oriented version of team splitting is the **splitall** loop:

```
splitall
  (int i=<start>; <cond>;
     i+= <step>[;[nteams][;tsize]])
   { <task> }
```

When a team encounters a **splitall** loop, it disassociates into subteams to which the indices of the loop are interleaved. The number and size of the subteams may be determined by the optional integer expressions, **nteams** (for specifying desired number of teams) and **tsize** (for desired size of teams), or by flags to PCP. If the appropriate number of processors is available at run-time, the user-supplied directives are followed. Otherwise, the number of teams and team size are determined by the implementation based on one or the other of the directives. The **splitall** task may be a function of the **splitall** index.

Summary

We have assembled ideas from several sources to create a parallel extension of ANSI C that can be used efficiently on a wide range of architectures. The design goal of the parallel programming model is to achieve reducibility on simpler architectural targets as we move up the evolutionary chain of architecture complexity. PCP is a relatively simple programming language that allows the user explicit control of both data loop parallelism and task parallelism via processor teams.

PCP is currently being used to implement the Accelerated Strategic Computing Initiative's (ASCI's) fast prototype code, EPIMETHEUS. EPIMETHEUS is a mixed physics code that combines piecewise parabolic method hydrodynamics and radiation transport algorithms.

For further information, please contact
Eugene Brooks, 510-423-7341,
brooks3@llnl.gov; or
Karen Warren, 510-422-9022,
kwarren@llnl.gov

Work performed under the auspices of the U.S. DOE by LLNL under contract No. W-7405-ENG-48